The OS, Processes, fork() & exec()

Computer Operating Systems, Fall 2023

Instructor: Travis McGaha

Head TAs: Nate Hoaglund & Seungmin Han

TAs:

Andy Jiang	Haoyun Qin	Kevin Bernat	Ryoma Harris
Audrey Yang	Jason hom	Leon Hertzberg	Shyam Mehta
August Fu	Jeff Yang	Maxi Liu	Tina Kokoshvili
Daniel Da	Jerry Wang	Ria Sharma	Zhiyan Lu
Ernest Ng	Jinghao Zhang	Rohan Verma	

Poll Everywhere

pollev.com/tqm

- How are you?

Administrivia

- Proj0 (penn-shredder) to be released soon (if not already)
  - This includes git & docker setup instructions. Do this part ASAP, it can take a while to debug issues with setup
  - **This assignment is done on your own**
- Check-in Quiz 0 to be released tonight or tomorrow
  - “Due” before lecture on Tuesday
  - Will keep open for a bit longer than that, to account for students joining the course a bit late

Poll Everywhere

pollev.com/tqm

- Any questions, comments or concerns?

Lecture Outline

- **Control Flow**
- Interrupts
- Processes
- fork()
- exec()

Control Flow

- Processors do only one thing:
  - From startup to shutdown, a CPU simply reads and executes (interprets) a sequence of instructions, one at a time
  - This sequence is the CPU's *control flow* (or *flow of control*)

Physical control flow

Time ↓

<startup>
$inst_1$
$inst_2$
$inst_3$
…
$inst_n$
<shutdown>

Poll Everywhere

pollev.com/tqm

The BRp instruction is being executed for the first time, which instruction is executed next?

```
        CONST R0, #5
        CONST R1, #2
        CONST R2, #0

LOOP    ADD R2, R2, #1
        SUB R0, R0, R1
        BRp LOOP
END     JMP #-1
```

- **A. BRp**
- **B. ADD**
- **C. SUB**
- **D. JMP**
- **E. I'm not sure**

Altering the Control Flow

- Up to now: two mechanisms for changing control flow:
  - Jumps and branches
  - Call and return

  React to changes in ***program state***

- Insufficient for a useful system:
  Difficult to react to changes in *system state*
  - Data arrives from a disk or a network adapter
  - Instruction divides by zero
  - User hits Ctrl-C at the keyboard
  - System timer expires

- System needs mechanisms for "exceptional control flow"

Exceptional Control Flow

- Exists at all levels of a computer system
- Low level mechanisms
  - 1. **Hardware Interrupts**
    - Change in control flow in response to a system event (i.e., change in system state)
    - Implemented using combination of hardware and OS software
- Higher level mechanisms
  - 2. **Process context switch**
    - Implemented by OS software and hardware timer
  - 3. **Signals**
    - Implemented by OS software

What we will be looking at today

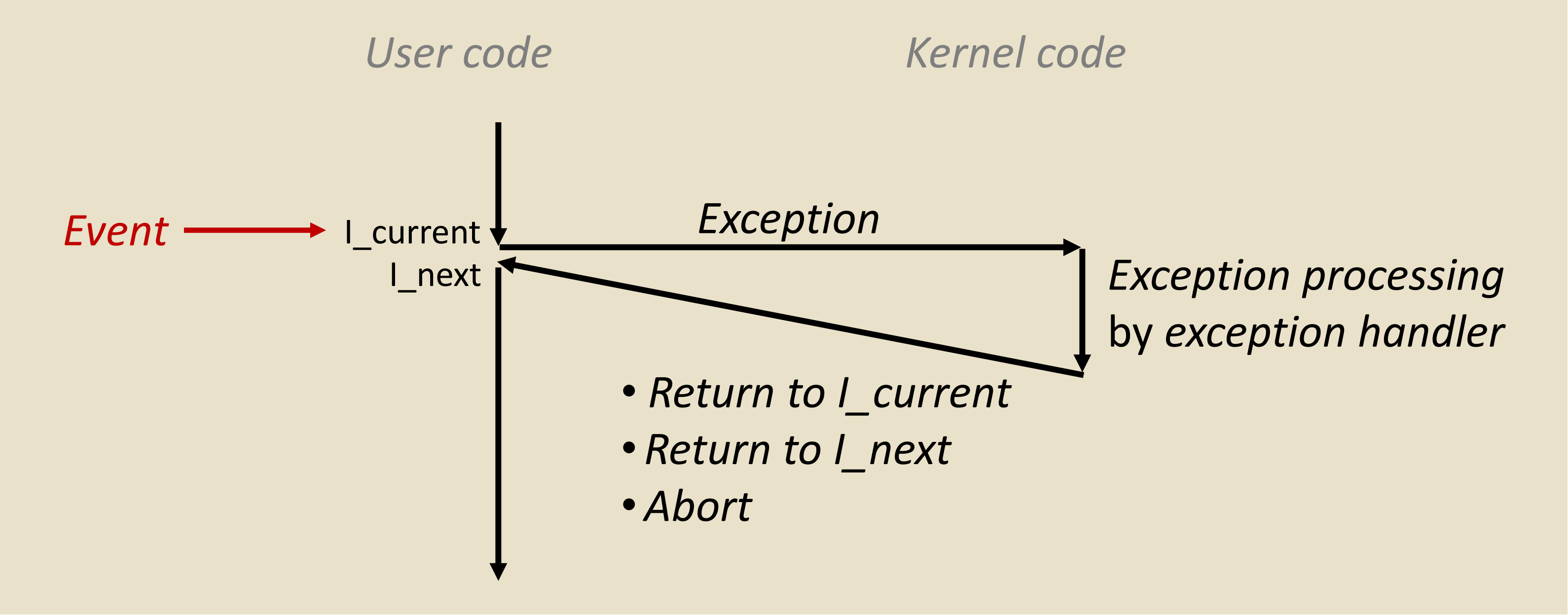
For next lecture

Lecture Outline

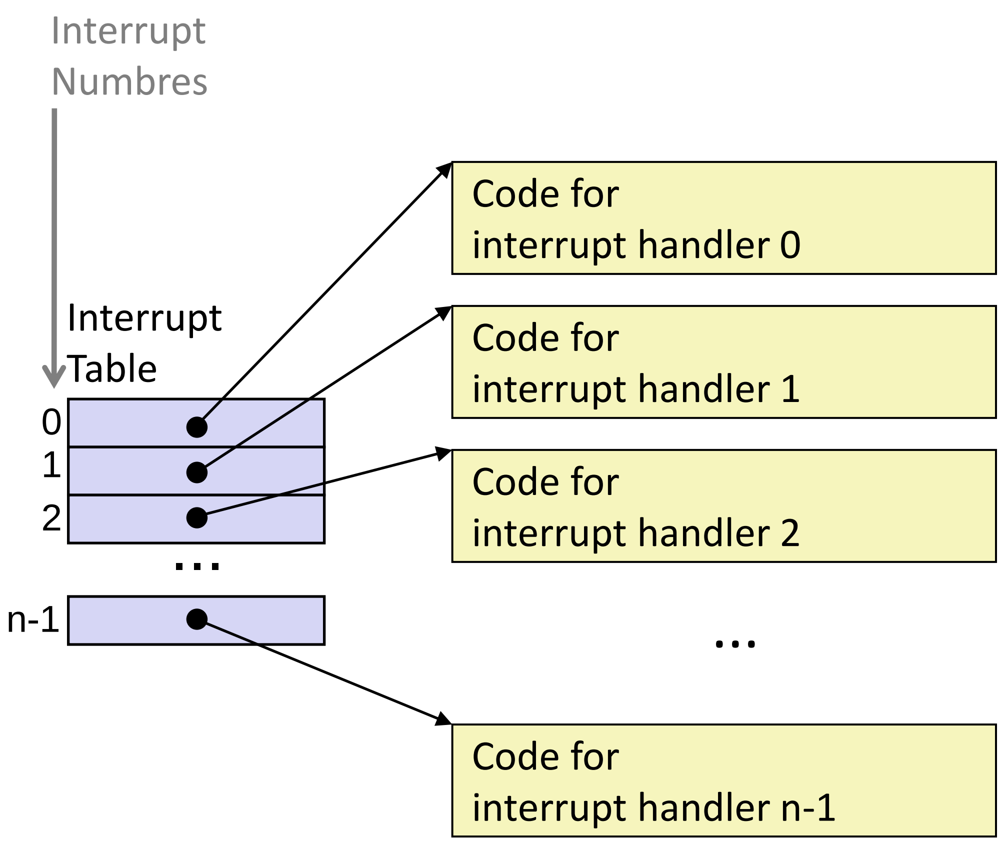
- Control Flow
- **Interrupts**
- Processes
- fork()
- exec()

Interrupts

- An *Interrupt* is a transfer of control to the OS *kernel* in response to some *event* (i.e., change in processor state)
  - Kernel is the memory-resident part of the OS
  - Examples of events: Divide by 0, arithmetic overflow, page fault, I/O request completes, typing Ctrl-C



Interrupt Tables



- Each type of event has a unique number k
- k = index into table (a.k.a. interrupt vector)
- Handler k is called each time interrupt k occurs

Asynchronous Interrupts

- Caused by events external to the processor
  - Indicated by setting the processor's *interrupt pin*
  - Handler returns to "next" instruction

- Examples:
  - Timer interrupt
    - Every few ms, an external timer chip triggers an interrupt
    - Used by the kernel to take back control from user programs
  - I/O interrupt from external device
    - Hitting Ctrl-C at the keyboard
    - Arrival of a packet from a network
    - Arrival of data from a disk

Synchronous Interrupts

- Caused by events that occur as a result of executing an instruction:
  - ***Traps***
    - Intentional
    - Examples: ***system calls***, breakpoint traps, special instructions
    - Returns control to "next" instruction
  - ***Faults***
    - Unintentional but theoretically recoverable
    - Examples: page faults (recoverable), protection faults (recoverable sometimes), floating point exceptions
    - Either re-executes faulting ("current") instruction or aborts
  - ***Aborts***
    - Unintentional and unrecoverable
    - Examples: illegal instruction, parity error, machine check
    - Aborts current program

FUN FACT: the terminology and definitions aren't fully agreed upon. Many people may use these interchangeably

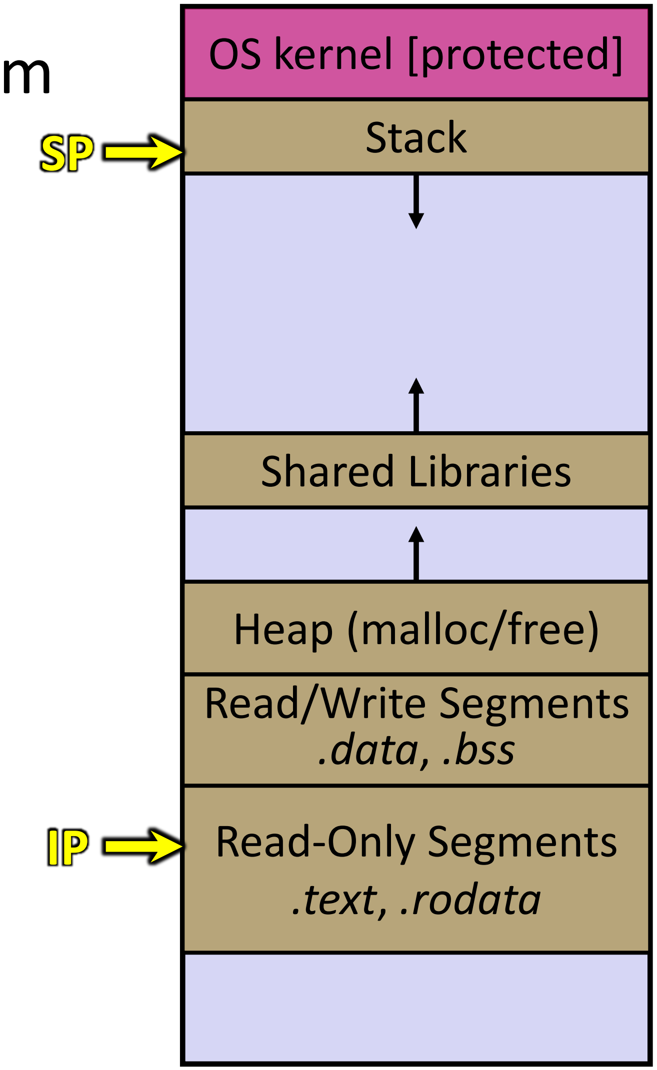
Lecture Outline

- Control Flow
- Interrupts
- Processes
- fork()
- exec()

Definition: Process

- Definition: An instance of a program that is being executed
(or is ready for execution)

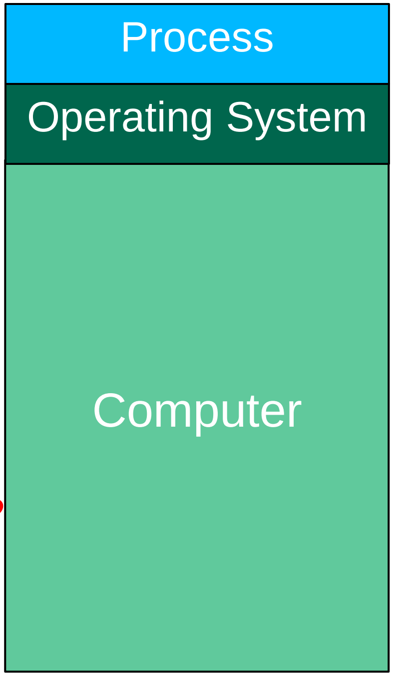
- Consists of:
  - Memory (code, heap, stack, etc)
  - Registers used to manage execution (stack pointer, program counter, ...)
  - Other resources



* This isn't quite true
more in a future lecture

Computers as we know them now

- In CIS 2400, you learned about hardware, transistors, CMOS, gates, etc.
- Once we got to programming, our computer looks something like:

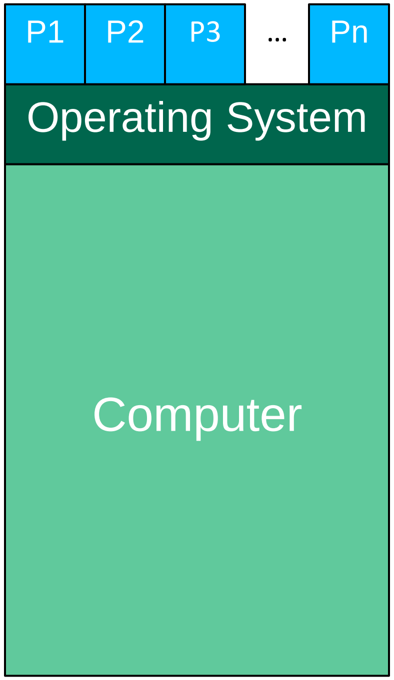


What is missing/wrong with this?

- This model is still useful, and can be used in many settings

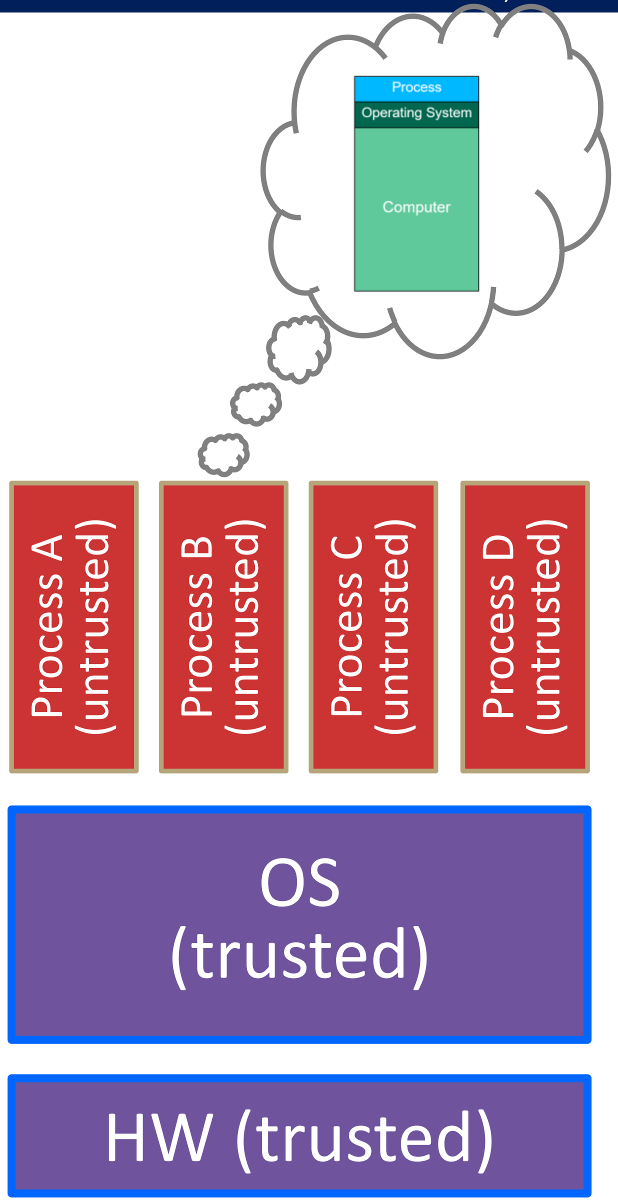
Multiple Processes

- Computers run multiple processes "at the same time"
- One or more processes for each of the programs on your computer
- Each process has its own…
  - Memory space
  - Registers
  - Resources

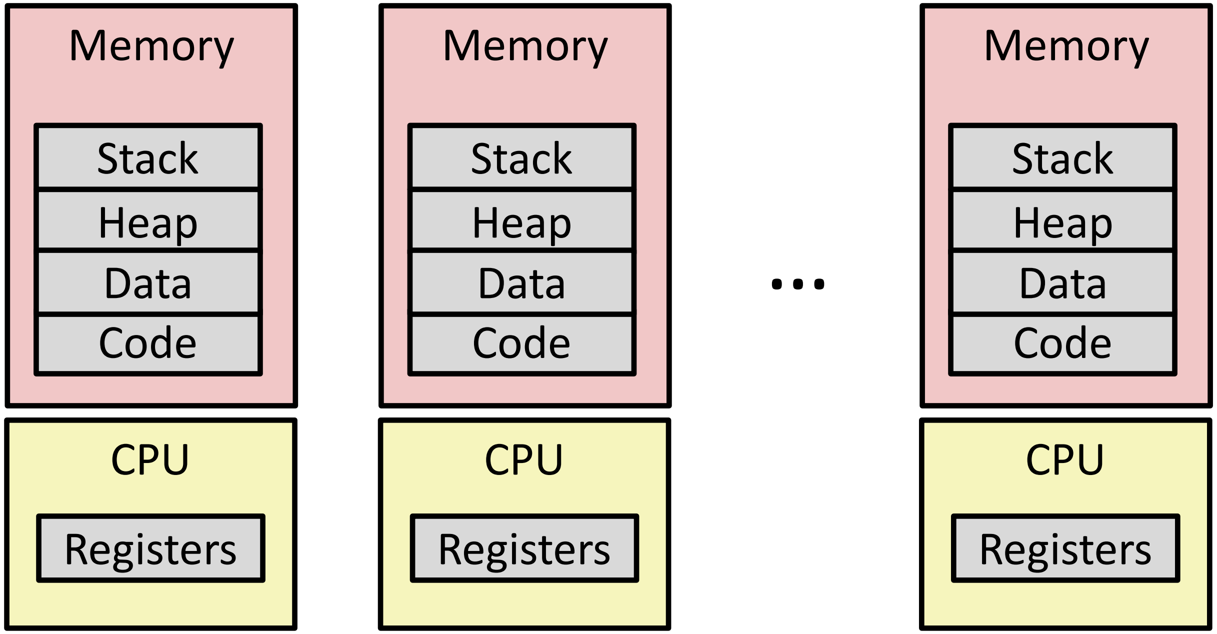


OS: Protection System

- OS isolates process from each other
  - Each process seems to have exclusive use of memory and the processor.
    - This is an **illusion**
    - More on Memory when we talk about virtual memory later in the course
  - OS permits controlled sharing between processes
    - E.g. through files, the network, etc.
- OS isolates itself from processes
  - Must prevent processes from accessing the hardware directly

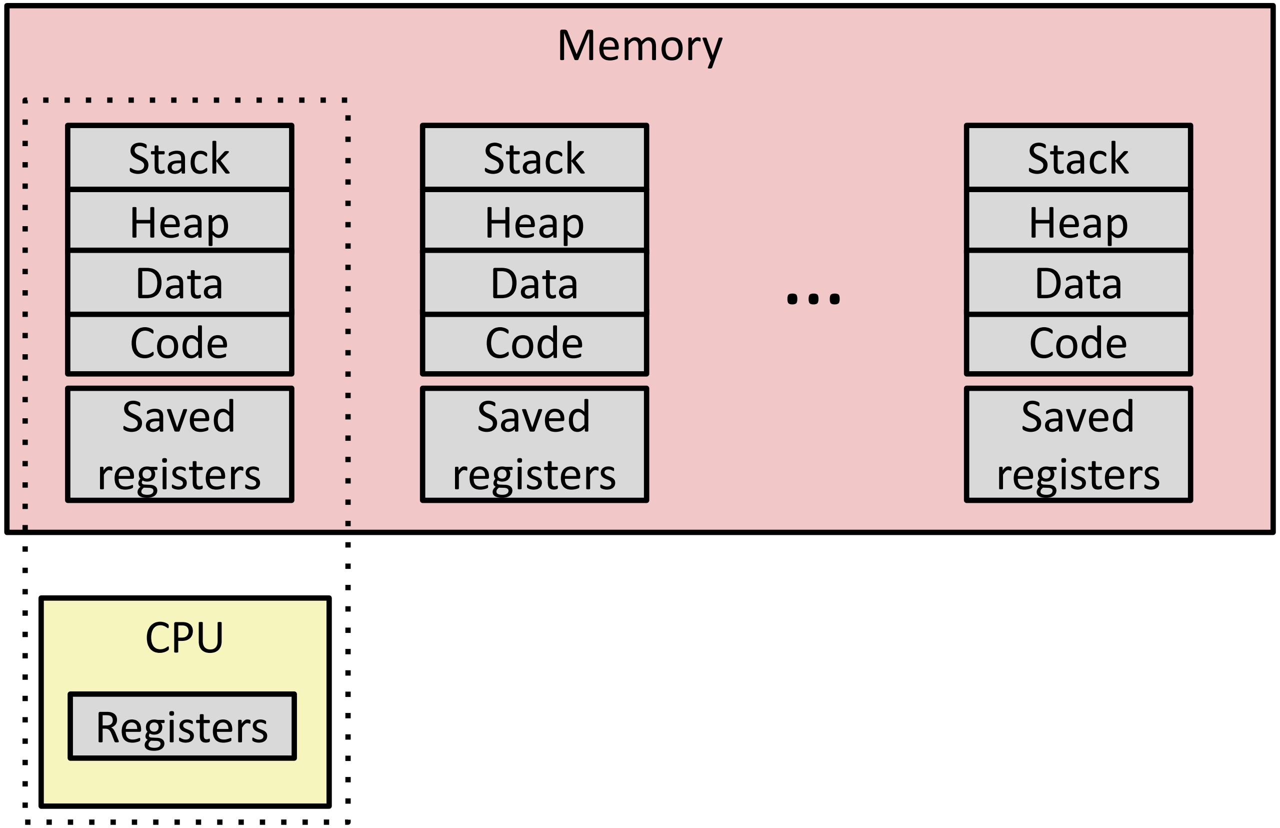


Multiprocessing: The Illusion



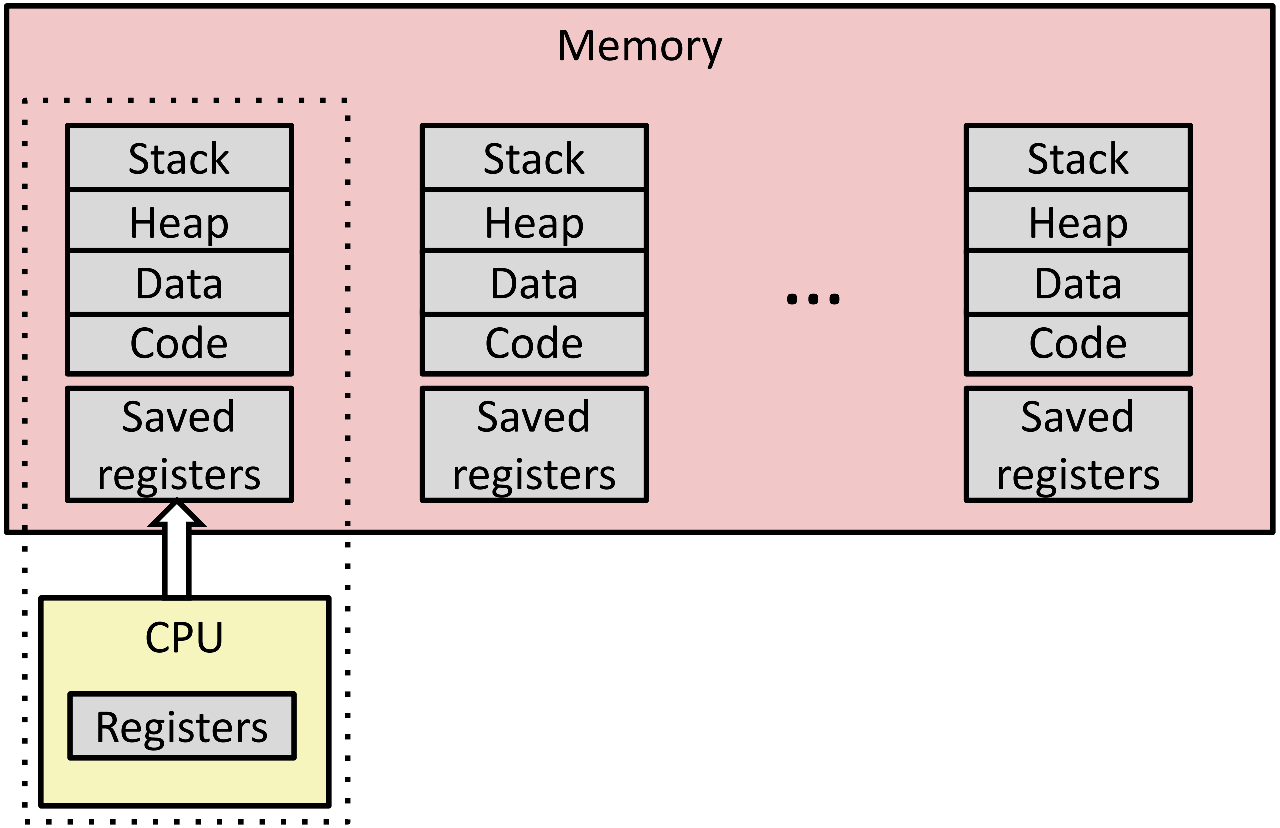
- Computer runs many processes simultaneously
  - Applications for one or more users
    - Web browsers, email clients, editors, …
  - Background tasks
    - Monitoring network & I/O devices

Multiprocessing: The (Traditional) Reality



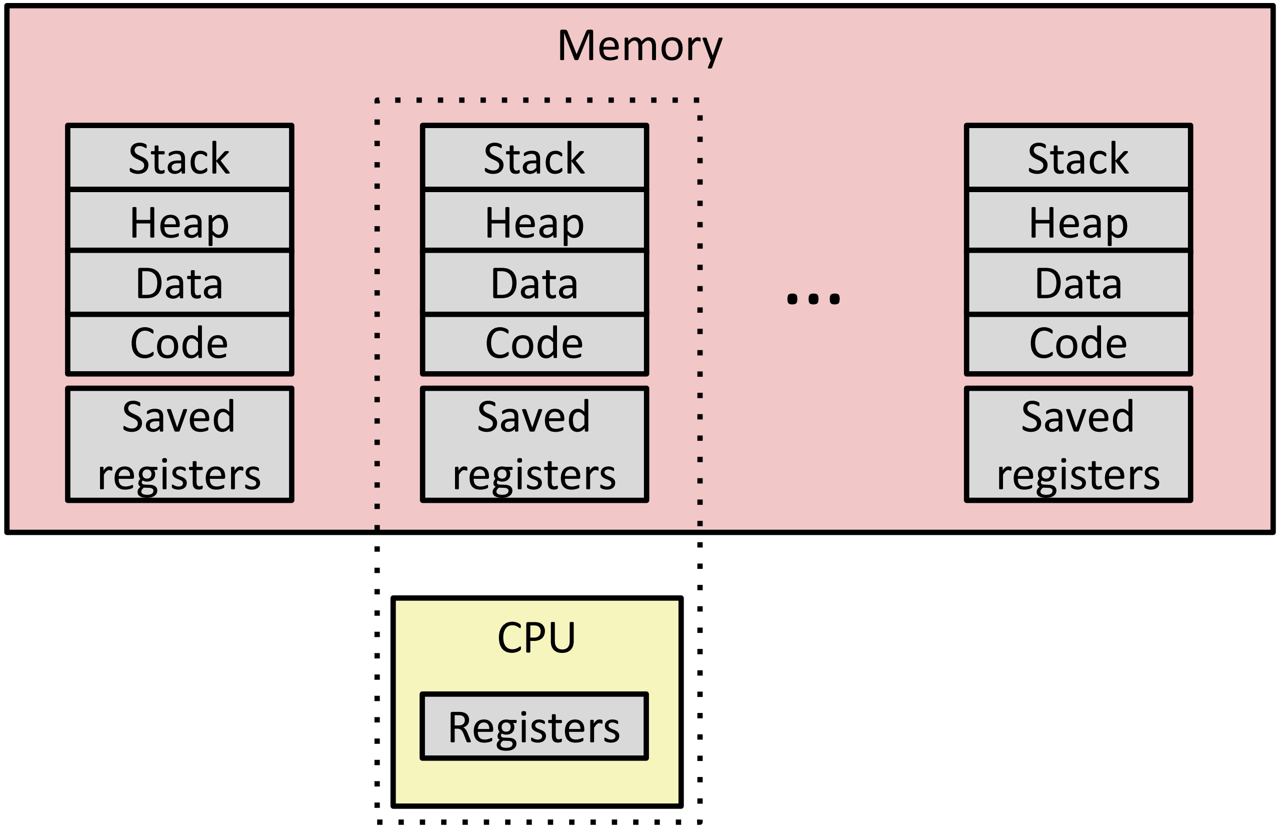
- Single processor executes multiple processes concurrently
  - Process executions interleaved (multitasking)
  - Address spaces managed by virtual memory system (later in course)
  - Register values for nonexecuting processes saved in memory

Multiprocessing: The (Traditional) Reality



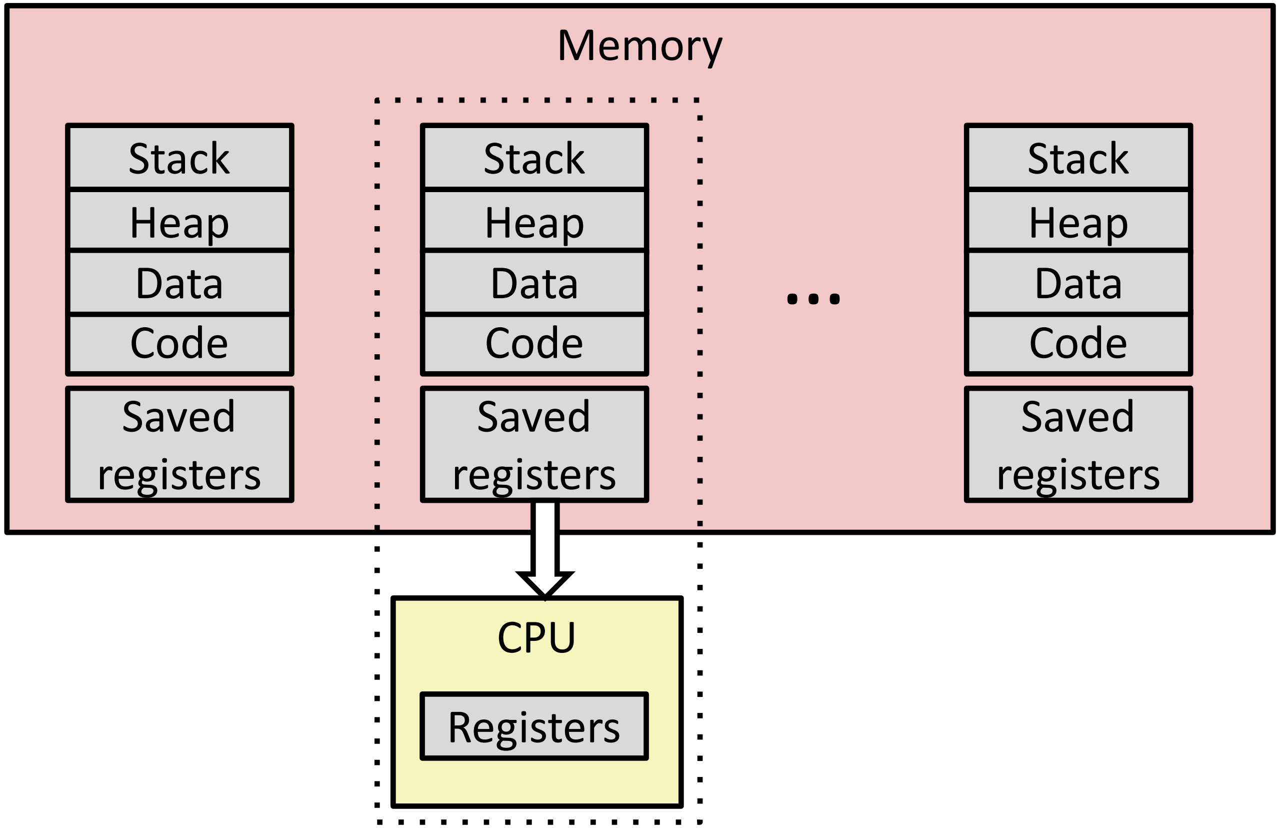
1. Save current registers in memory

Multiprocessing: The (Traditional) Reality



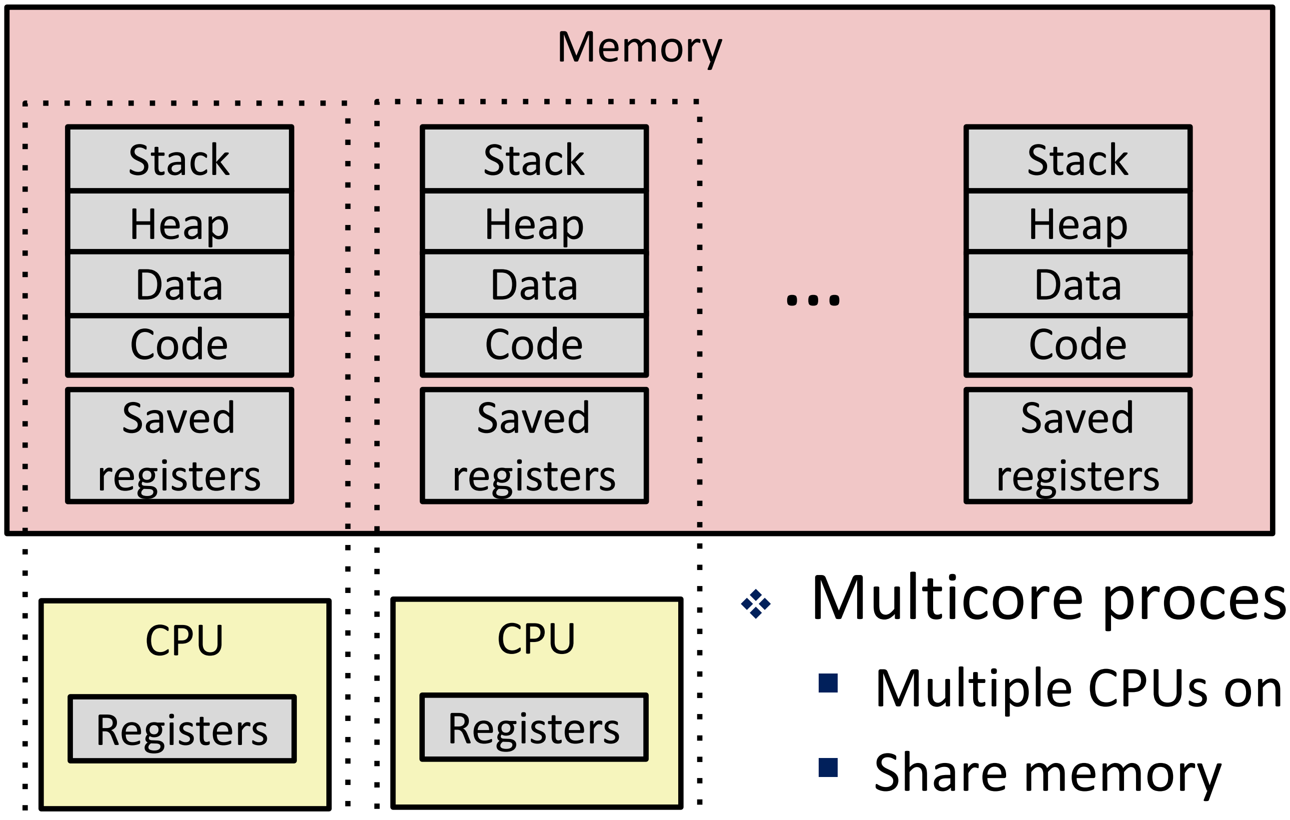
1. Save current registers in memory
2. Schedule next process for execution

Multiprocessing: The (Traditional) Reality



1. Save current registers in memory
2. Schedule next process for execution
3. Load saved registers and switch address space (context switch)

Multiprocessing: The (Modern) Reality



- Multicore processors
  - Multiple CPUs on single chip
  - Share memory
  - Each can execute a separate process
    - Scheduling of processors onto cores done by kernel
  - This is called “Parallelism”

Poll Everywhere

pollev.com/tqm

- What I just went through was the big picture of processes. Many details left, some will be gone over in future lectures

- Any questions, comments or concerns so far?

Process States (incomplete)

FOR NOW, we can think of a process as being in one of three states:

More states in future lectures

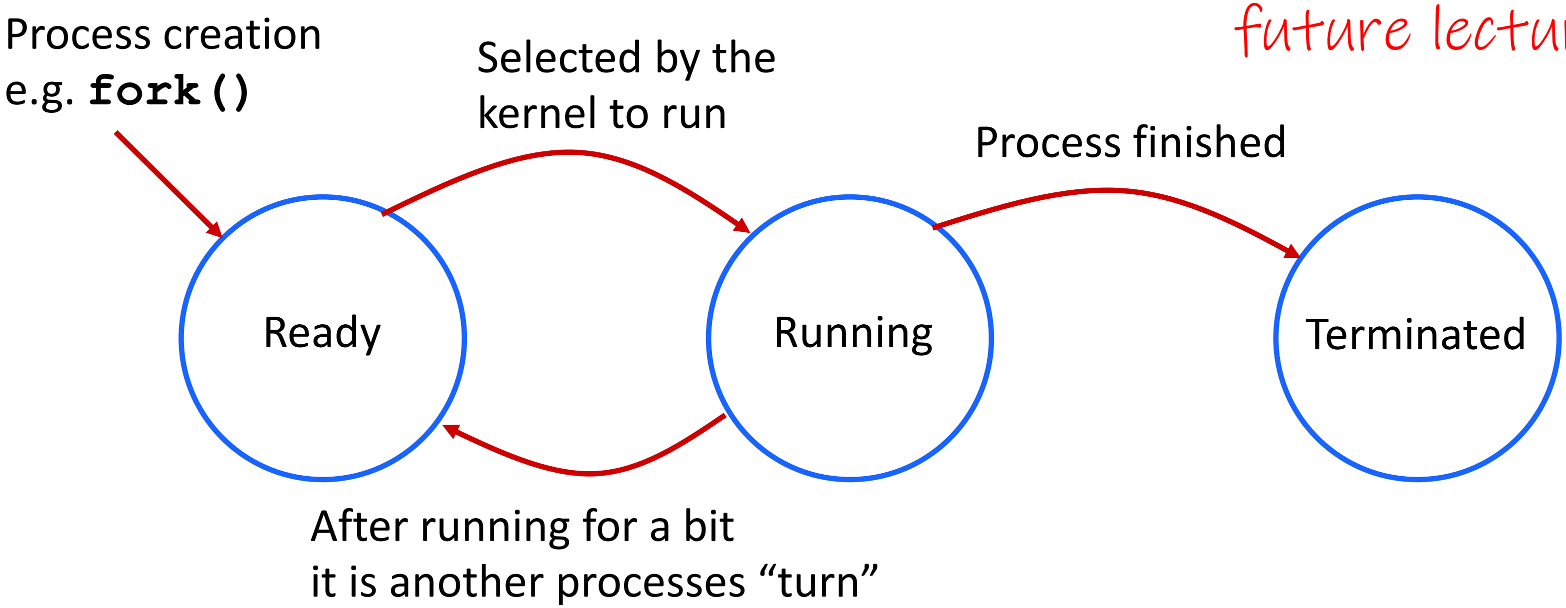
- Running
  - Process is currently executing
- Ready
  - Process is waiting to be executed and will eventually be *scheduled* (i.e., chosen to execute) by the kernel

Scheduler to be covered in a later lecture

- Terminated
  - Process is stopped permanently

Process State Lifetime (incomplete)

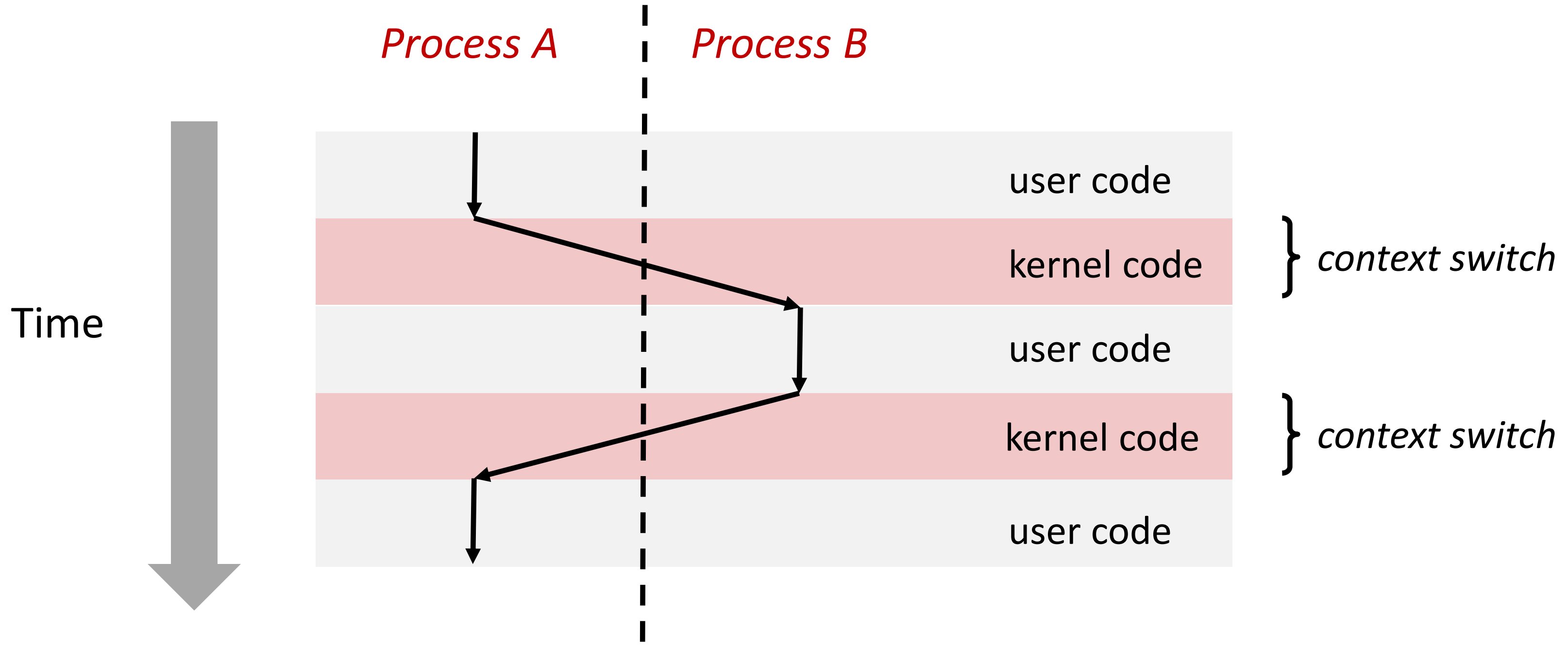
More states in future lectures



Processes can be “interrupted” to stop running. Through something like a hardware timer interrupt

Context Switching

- Processes are managed by a shared chunk of memory-resident OS code called the *kernel*
  - Important: the kernel is not a separate process, but rather runs as part of some existing process.
- Control flow passes from one process to another via a *context switch*



OS: The Scheduler

- When switching between processes, the OS will run some kernel code called the “Scheduler”
- The scheduler runs when a process:
  - starts (“arrives to be scheduled”),
  - Finishes
  - Blocks (e.g., waiting on something, usually some form of I/O)
  - Has run for a certain amount of time

- It is responsible for scheduling processes
  - Choosing which one to run
  - Deciding how long to run it

Scheduler Considerations

- The scheduler has a scheduling algorithm to decide what runs next.
- Algorithms are designed to consider many factors:
  - Fairness: Every program gets to run
  - Liveness: That “something” will eventually happen
  - Throughput: Number of “tasks” completed over an interval of time
  - Wait time: Average time a “task” is “alive” but not running
  - A lot more...
- More on this later. **For now: think of scheduling as non-deterministic**, details handled by the OS.

Lecture Outline

- Control Flow
- Interrupts
- Processes
- **fork()**
- exec()

Terminating Processes

- Process becomes terminated for one of three reasons:
  - Receiving a signal whose default action is to terminate (next lecture)
  - Returning from the `main` routine
  - Calling the `exit` function

- `void exit(int status);`
  - Terminates with an *exit status* of `status`
  - Convention: normal return status is 0, nonzero on error
  - Another way to explicitly set the exit status is to return an integer value from the main routine

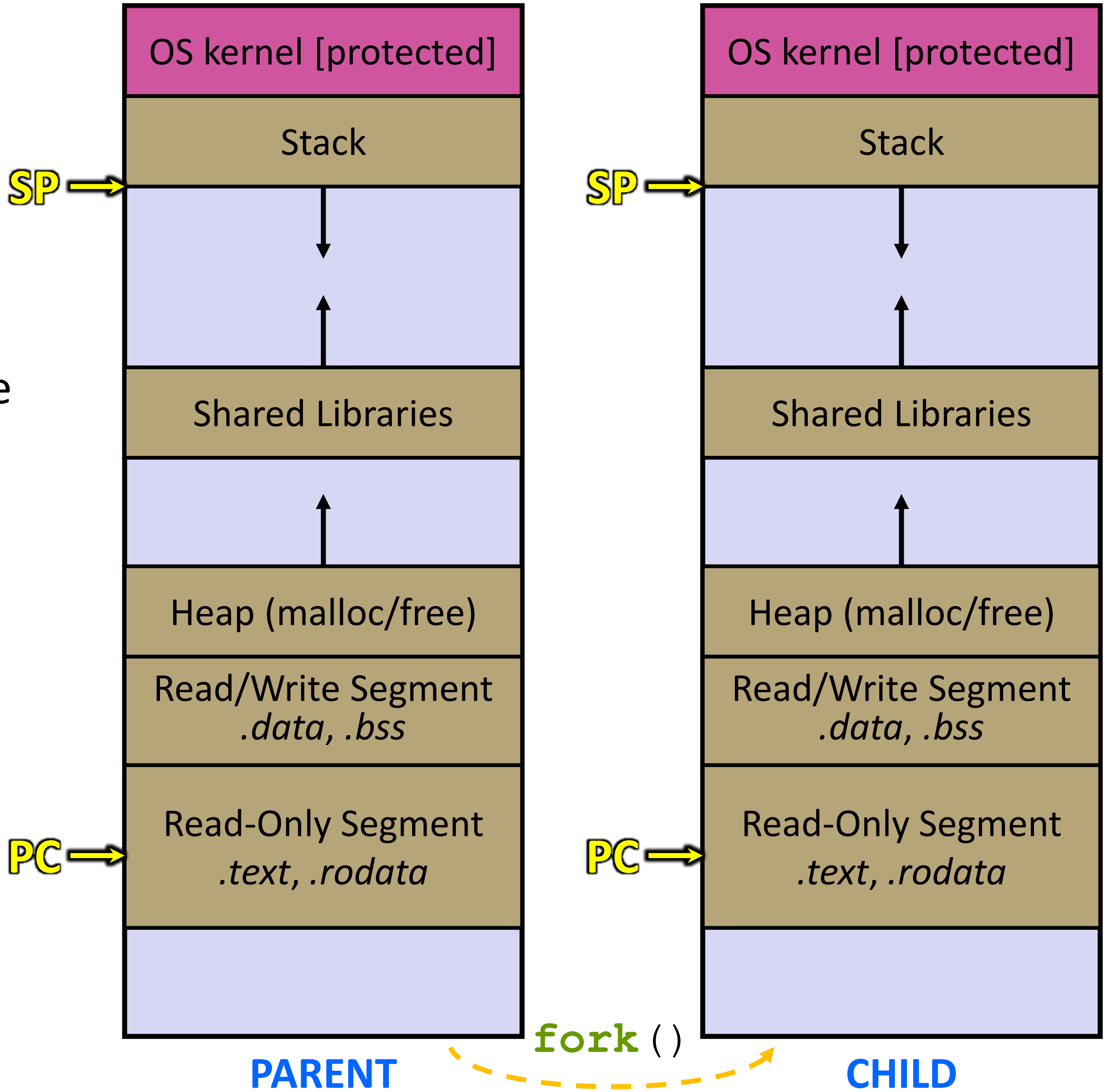
- `exit` is called once but never returns.

Creating New Processes

- `pid_t fork();`
  - Creates a new process (the "child") that is an *exact clone** of the current process (the "parent")
    - *almost everything
  - The new process has a separate virtual address space from the parent
  - Returns a **`pid_t`** which is an integer type.

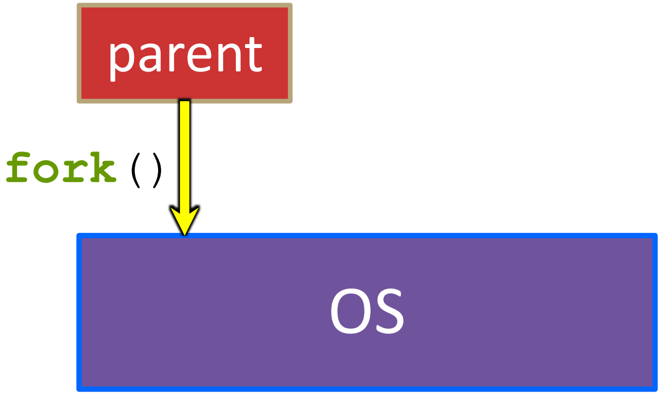
`fork()` and Address Spaces

- Fork causes the OS to clone the address space
  - The *copies* of the memory segments are (nearly) identical
  - The new process has *copies* of the parent's data, stack-allocated variables, open file descriptors, etc.



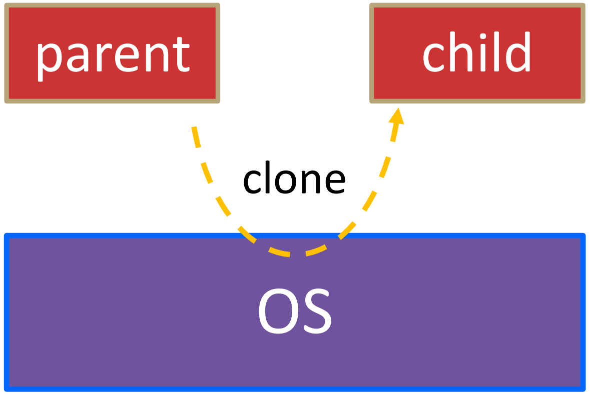
fork()

- fork() has peculiar semantics
  - The parent invokes fork()
  - The OS clones the parent
  - *Both* the parent and the child return from fork
    - Parent receives child's pid
    - Child receives a 0



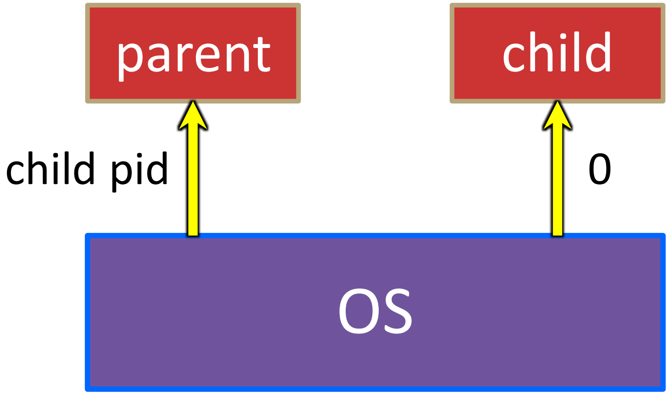
fork()

- `fork()` has peculiar semantics
  - The parent invokes `fork()`
  - The OS clones the parent
  - *Both* the parent and the child return from fork
    - Parent receives child's pid
    - Child receives a 0



`fork()`

- `fork()` has peculiar semantics
  - The parent invokes `fork()`
  - The OS clones the parent
  - *Both* the parent and the child return from fork
    - Parent receives child's pid
    - Child receives a 0



"simple" `fork()` example

```
fork();
printf("Hello!\n");
```

- What does this print?

"simple" fork() example

```
int x = 3;
fork();
x++;
printf("%d\n", x);
```

- What does this print?

fork() example

```
pid_t fork_ret = fork();

if (fork_ret == 0) {
  printf("Child\n");
} else {
  printf("Parent\n");
}
```

fork() example

Parent Process (PID = X)

```
pid_t fork_ret = fork();

if (fork_ret == 0) {
  printf("Child\n");
} else {
  printf("Parent\n");
}
```

Child Process (PID = Y)

```
pid_t fork_ret = fork();

if (fork_ret == 0) {
  printf("Child\n");
} else {
  printf("Parent\n");
}
```

fork()

fork() example

Parent Process (PID = X)

```
pid_t fork_ret = fork();

if (fork_ret == 0) {
  printf("Child\n");
} else {
  printf("Parent\n");
}
```

Child Process (PID = Y)

```
pid_t fork_ret = fork();

if (fork_ret == 0) {
  printf("Child\n");
} else {
  printf("Parent\n");
}
```

fork_ret = Y

```
pid_t fork_ret = fork();

if (fork_ret == 0) {
  printf("Child\n");
} else {
  printf("Parent\n");
}
```

Prints "Parent"

fork_ret = 0

```
pid_t fork_ret = fork();

if (fork_ret == 0) {
  printf("Child\n");
} else {
  printf("Parent\n");
}
```

Prints "Child"

Which prints first?

Non-deterministic

Another fork() example

```
pid_t fork_ret = fork();
int x;

if (fork_ret == 0) {
  x = 3800;
} else {
  x = 2400;
}
printf("%d\n", x);
```

Another fork() example

Parent Process (PID = X)

```
pid_t fork_ret = fork();
int x;

if (fork_ret == 0) {
  x = 3800;
} else {
  x = 2400;
}
printf("%d\n", x);
```

Child Process (PID = Y)

```
pid_t fork_ret = fork();
int x;

if (fork_ret == 0) {
  x = 3800;
} else {
  x = 2400;
}
printf("%d\n", x);
```

fork()

Another fork() example

Parent Process (PID = X)

```
pid_t fork_ret = fork();
int x;

if (fork_ret == 0) {
  x = 3800;
} else {
  x = 2400;
}
printf("%d\n", x);
```

Child Process (PID = Y)

```
pid_t fork_ret = fork();
int x;

if (fork_ret == 0) {
  x = 3800;
} else {
  x = 2400;
}
printf("%d\n", x);
```

fork_ret = Y

fork()

fork_ret = 0

Always prints "2400"

Always prints "3800"

Reminder: Processes have their own address space (and thus, copies of their own variables)

Order is still nondeterministic!!

Lecture Outline

- Control Flow
- Interrupts
- Processes
- fork()
- **exec()**

exec*()

- Loads in a new program for execution

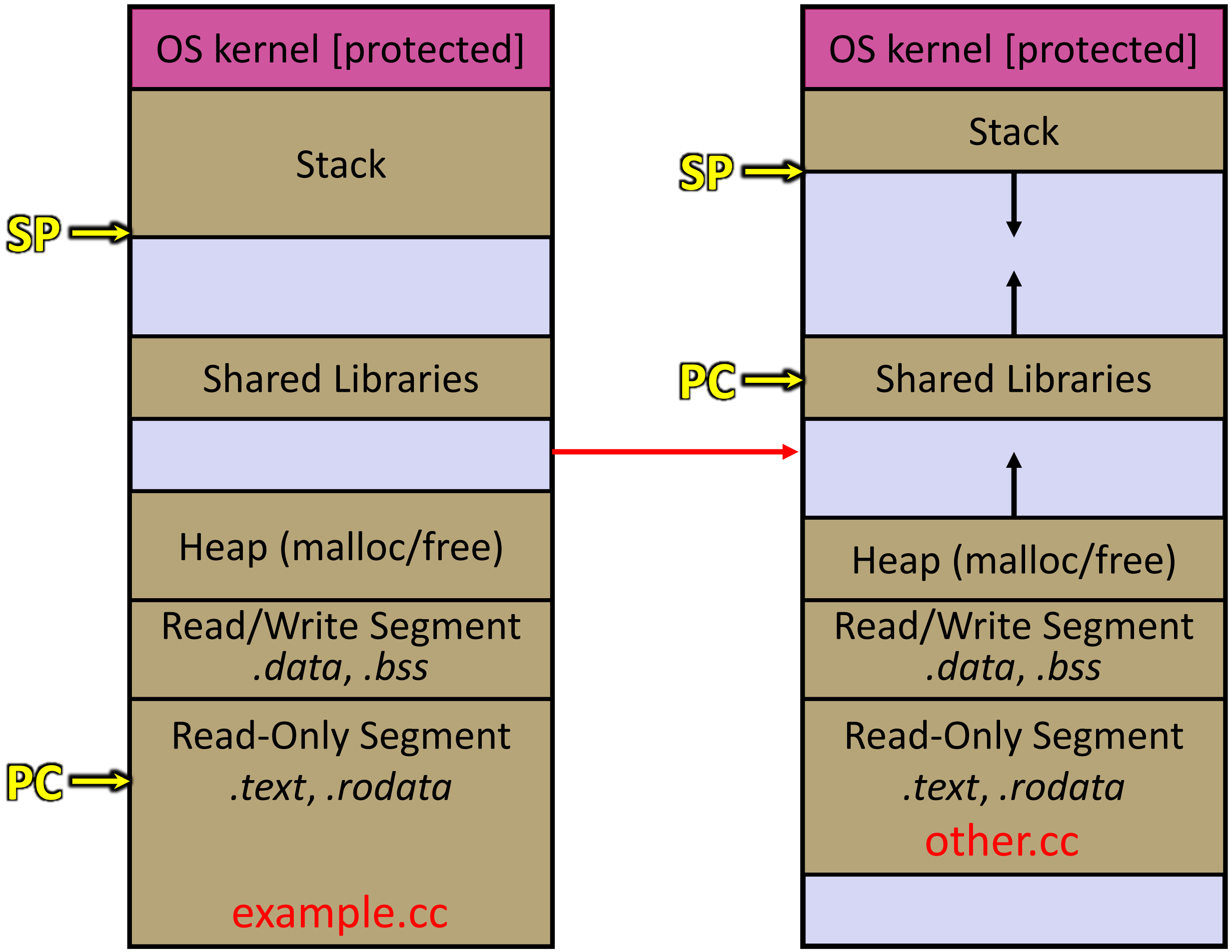
- PC, SP, registers, and memory are all reset so that the specified program can run

execve()

- ```
  int execve(const char *file,
             char* const argv[]
             char* const envp[]);
  ```
- Duplicates the action of the shell (terminal) in terms of finding the command/program to run
- Argv is an array of **`char*`**, the same kind of argv that is passed to `main()` in a C program
  - **`argv[0]`** MUST have the same contents as the file parameter
  - **`argv`** must have NULL as the last entry of the array
- Just pass in an array of `{ NULL };` as envp
- Returns -1 on error. Does NOT return on success
  ```

Exec Visualization

- Exec takes a process and discards or "resets" most of it



NOTE that the following DO change

- The stack
- The heap
- Globals
- Loaded code
- Registers

NOTE that the following do NOT change

- Process ID
- Open files
- The kernel

Exec Demo

- See `exec_example.c`
  - Brief code demo to see how exec works
  - What happens when we call exec?
  - What happens to allocated memory when we call exec?

Poll Everywhere

pollev.com/tqm

```
int main(int argc, char* argv[]) {
  char* envp[] = { NULL };
  // fork a process to exec clang
  pid_t clang_pid = fork();
  if (clang_pid == 0) {
    // we are the child
    char* clang_argv[] = {"/bin/clang", "-o",
                  "hello","hello_world.c", NULL};
    execve(clang_argv[0], clang_argv, envp);
    exit(EXIT_FAILURE);
  }

  // fork to run the compiled program
  pid_t hello_pid = fork();
  if (hello_pid == 0) {
    // the process created by fork
    char* hello_argv[] = {"./hello", NULL};
    execve(hello_argv[0], hello_argv, envp);
    exit(EXIT_FAILURE);
  }
  return EXIT_SUCCESS;
}
```

broken_autograder.c

This code is broken. It compiles, but it doesn't do what we want. Why?

- Clang is a C compiler
- Assume it compiles
- Assume I gave the correct args to exec